IRAM Memo 2013-3

CLASSIC

Application Programming Interface

S. Bardeau[1], V. Pietu[1], J. Pety[1,2]

1. IRAM (Grenoble)
2. LERMA, Observatoire de Paris

March, 18^{th} 2015
Version 1.0

Abstract

We describe in details the Application Programming Interface of the **CLASSIC** Library, provided by **GILDAS**.

Contents

1 Description

GILDAS provides its own library (Application Programming Interface with subroutines, Fortran derived types, and parameters) which is able to read and write files using the **CLASSIC** Data Container. The idea is to provide a modernized API to access the **CLASSIC** Data Container from **CLASS** and **CLIC** (historical applications) and open access to new ones (*e.g.* **MRTCAL**). The library does not use internally any global variable (keeping in mind possible parallel processing in the future), which becomes possible thanks to the Fortran derived types.

This API is not intended to be public (in a first time at least), but just to be used by **GILDAS** programs using the **CLASSIC** Data Container, namely **CLASS**, **CLIC**, and **MRTCAL**. It is presented in details in Appendix A.

1.1 Support of old files

Table 1: Backward compatibility of old systems or files.

Kind	IEEE	EEEI	VAX	PDP-11
V1 multiple	RW	RW	RW	No
V1 single	RW	RW	RW?	No
V2	RW	RW	RW?	No

The **CLASSIC** Library supports reading and writing both Version 1 and Version 2 files, transparently for the calling application. The old PDP-11 files (with Integer*2 words) are not supported since Class90 (and Class77 only supported reading). VAX is supported for reading; writing is available but not tested since no VAX systems can easily be found nowadays.

2 Application Programming Interface

This section describes the public procedures and Fortran derived types. The calling application *must* use the module `classic_api` in order to use them. In particular procedures may invoke Fortran-90 and above calling sequences which require a so-called *explicit interface* by the Fortran standard. The other procedures or types elements which are not detailed here are not public and are subject to change without advertisement.

In the derived types described below, some elements are Read-Write (RW), Read-Only (RO) or PRivate (PR, understand *invisible*) for the calling application. Private elements existence, name, and definition are subject to change without further notice.

2.1 Codes and parameters

A few parameters are available for use by the applications:

- `integer, parameter :: entry_length=8`: the kind for all integers describing an entry number (or number of entries). Some of the **CLASSIC** public subroutines expect such integers.
- `integer, parameter :: data_length=8`: the kind for all integers describing the number of words in the `data` block and the section lengths.
- `integer, parameter :: classic_reclen_v1=128`: the record length (words) in V1 files is provided for programmer convenience and better readability of the code.
- `integer, parameter :: classic_kind_demo=-1`,

- `integer, parameter :: classic_kind_unknown=0`,
- `integer, parameter :: classic_kind_class=1`,
- `integer, parameter :: classic_kind_clic=2`,
- `integer, parameter :: classic_kind_mrtcal=3`: provide the code for the file kind, *i.e.* each application can check if it can read the given file or not. The code `classic_kind_demo` can be used for demonstrative Classic files, for testing purpose only. The code `classic_kind_unknown` will be returned when reading V1 files; it can not be used when writing a new file.

2.2 File API

```
type classic_file_t
   character(len=256)       :: spec=''    ! RW - File name
   integer(kind=4)          :: nspec=0    ! RW - Filename length
   integer(kind=4)          :: lun=0      ! RW - Logical unit
   logical                  :: readwrite  ! PR - Read-only or Read-Write?
   type(classic_filedesc_t) :: desc       ! RW - File descriptor
   logical                  :: update     ! RW - Is file opened for Update ?
   type(classic_fileconv_t) :: conv       ! RO - Conversion routines
end type classic_file_t
```

This high-level type provides a convenient way to describe a **CLASSIC** file. Associated subroutines are:

- `classic_file_init`: Initialize a new Classic file
- `classic_file_open`: Open an old Classic file
- `classic_file_close`: Close a Classic file
- `classic_file_loss`: Parse the input file and search for unused (lost) bytes (debugging printouts)
- `classic_file_copy`: Copy a type(classic_file_t) into another one
- `classic_file_fopen`: Fortran-open the given Classic file
- `classic_file_fclose`: Fortran-close the given Classic file
- `classic_file_fflush`: Fortran-flush the given Classic file

2.3 File Descriptor API

```
type classic_filedesc_t
  sequence  ! Not really needed since we always access these elements one by one
  integer(kind=4)              :: code     ! PR - 1             File code
  integer(kind=4)              :: reclen   ! RO - 2             Record length
  integer(kind=4)              :: kind     ! RO - 3             File kind
  integer(kind=4)              :: vind     ! RO - 4             Index version
  integer(kind=4)              :: lind     ! PR - 5             Index length
  integer(kind=4)              :: flags    ! PR - 6 (bit #1)    Single or multiple?
                                           ! PR - 6 (bits 2:32) Provision (0-filled)
  integer(kind=8)              :: xnext    ! RW - 7:8           Next available entry number
  integer(kind=8)              :: nextrec  ! RO - 9:10          Next record which contains free space
  integer(kind=4)              :: nextword ! RO - 11            Next available word in this record
```

```
  integer(kind=4)              :: lex1     ! PR - 12           Length of first extension (number of en
  integer(kind=4)              :: nex      ! PR - 13           Number of extensions
  integer(kind=4)              :: gex      ! PR - 14           Extension growth rule
  integer(kind=8), allocatable :: aex(:)   ! PR - 15:reclen    Extension addresses
  ! Not in data:
  integer(kind=4)              :: version  ! RO - 1 or 2  (duplicate of file code)
  logical                      :: single   ! RO - Obs numbering mode (duplicate of 'flags', bit #1)
  integer(kind=4)              :: mex      ! PR - Size of aex(:) and lexN(:)
  integer(kind=4)              :: pad1     !      Memory padding
  integer(kind=8), allocatable :: lexN(:)  ! PR - Extension lengths (cumulative, number of entries)
end type classic_filedesc_t
```

Note that some elements (marked PRivate) of the **CLASSIC** Data Container can be recognized but shall not be accessed by the application. For example the elements related to the extensions are fully handled by the **CLASSIC** Library. For some others (e.g. `code` or `flags` array) a convenient equivalent is provided (resp. `version` or `single`).

- `classic_filedesc_init`: Initialize a new File Index,
- `classic_filedesc_open`: Open and read the File Index,
- `classic_filedesc_read`: Read the File Index,
- `classic_filedesc_write`: Write the File Index,
- `classic_filedesc_dump`: Dump the content of the File Index.

The **CLASSIC** Library does not support extensions growth other than $gex = 10$ and $gex = 20$. If it appears in the future that $gex = 20$ is a too strong growth, intermediate values will be implemented. Note also that the library makes special efforts for the linear growth to remain fast in this simple case (*i.e.* generic formulas are not used).

In case of linear growth ($gex = 10$), the extension length (`lex1`, same for all extensions) is usually derived from the total number of entries desired per file, such as all the entries are uniformely divided in the `nex` extensions. The resulting value is rounded to the lower multiple of int($reclen/lind$) (integer division), *i.e.* the records devoted to the File Index are used at best.

In case of exponential growth ($gex = 20$), $lex1$ is always set to int($1 * reclen/lind$), *i.e.* the first Extension Index uses 1 record only. The **CLASSIC** Data Container allows the first Extension Index to be larger, but the **CLASSIC** Library does not offer the possibility to customize it.

2.4 File Index API

One of the strength of the **CLASSIC** Data Container is its ability to provide a summary (short and quickly accessible) of each entry. Namely: the File Index. Thus the **CLASSIC** Library provides a specific API to access the Entry Indexes, independently from the actual entry. Remember these elements are under the control of the calling application, except the rules detailed in the **CLASSIC** Data Container memo.

- `classic_entryindex_read`: Read the Entry Index of an entry,
- `classic_entryindex_write`: Write or replace the Entry Index.

2.5 Entry API

The **CLASSIC** Library does not provide any Fortran derived type associated to an entry, since it is private to each application. The subroutines below just read/write bytes from/to a buffer (an array of any type of 4-byte words that **CLASSIC** just uses without knowing their meaning) provided by the application.

- `classic_entry_init`: Initialize a new entry
- `classic_entry_section_read`: Read a section from an entry
- `classic_entry_data_read`: Read the 'data' block from an entry
- `classic_entry_data_readsub`: Read a subset of the 'data' block from an entry
- `classic_entry_section_add`: Add a new section to an entry
- `classic_entry_section_update`: Update an old section in an entry
- `classic_entry_data_add`: Write the 'data' block to the entry
- `classic_entry_data_update`: Update the 'data' block to the entry
- `classic_entry_close`: Finalize the entry writing

2.6 Entry Descriptor API

```
type classic_entrydesc_t
  sequence
  integer(kind=4) :: code     ! PR -  1    Code observation icode
  integer(kind=4) :: version  ! RO -  2    Observation version
  integer(kind=4) :: nsec     ! PR -  3    Number of sections
  integer(kind=4) :: pad1     !            Memory padding (not in data)
  integer(kind=8) :: nword    ! PR -  4: 5 Number of words
  integer(kind=8) :: adata    ! PR -  6: 7 Data address
  integer(kind=8) :: ldata    ! RO -  8: 9 Data length
  integer(kind=8) :: xnum     ! RO - 10:11 Entry number
  ! Out of the 'sequence' block:
  integer(kind=4) :: msec     ! PR - Not in data: maximum number of sections the
                              !      Observation Index can hold
  integer(kind=4) :: pad2     !      Memory padding for 8 bytes alignment
  integer(kind=4) :: seciden(classic_maxsec)  ! PR - Section Numbers (on disk: 1 to ed%nsec)
  integer(kind=8) :: secleng(classic_maxsec)  ! RO - Section Lengths (on disk: 1 to ed%nsec)
  integer(kind=8) :: secaddr(classic_maxsec)  ! PR - Section Addresses (on disk: 1 to ed%nsec)
end type classic_entrydesc_t
```

- `classic_entrydesc_read`: Read an Entry Descriptor,
- `classic_entrydesc_write`: Write an Entry Descriptor,
- `classic_entrydesc_secfind_one`: Find if one section is present in the Entry Descriptor,
- `classic_entrydesc_secfind_all`: Find all sections present in the Entry Descriptor,
- `classic_entrydesc_dump`: Dump the content of the Entry Descriptor.

2.7 Buffer API

The **CLASSIC** Library provides a *record buffer* type. It is aimed to map (duplicate in memory) an actual record of a **CLASSIC** file. The idea behind this is that there are many times where the library or its user wants to access repeatedly elements which are in the same record. Instead of accessing it directly in the file (Fortran `read`/`write` file access by record), it is accessed in memory instead.

The associated Fortran derived type is the following:

```
type classic_recordbuf_t
  ! Object descriptor
  integer(kind=8) :: rstart  ! RO - The record where the object starts
  integer(kind=4) :: wstart  ! RO - In this record, the word where the object starts
  integer(kind=8) :: nrec    ! PR - Number of records used by the object
  ! Reading buffer
  integer(kind=4)              :: lun      ! RO - Associated file unit
  integer(kind=8)              :: roff     ! PR - Offset record from reference
  integer(kind=8)              :: len      ! PR - Size of data
  integer(kind=4), allocatable :: data(:)  ! RW - The buffer
end type classic_recordbuf_t
```

Note that this type describes more than a duplicated record in memory. It is though to be used to read/write any kind of object which data can be found in one or more records. The associated subroutines reflects this feature. All the benefits of the buffering will then make sense when successively accessing objects contiguous on the file, since the subroutines will avoid re-buffering a record already put in memory for a previous object.

There are some good programming practices for best use of the record buffers:

- the record is buffered in the `data(:)` (allocatable) array (the record and this array have the same size). The consequence of this is that several **CLASSIC** files can not share the same *record buffers*, since the record length and thus the data array may vary from one file to another.
- since the File Index and Entries themselves can not share pieces of the same records (Extension Indexes are aligned on their own records), it seems much cleverer to use one buffer for the entry indexes, and one buffer for the entries.
- there should be only one set of writing buffers. If it happens that several buffers duplicate the same record, modifications made in one would be lost when writing the other.
- on the other hand, there is no restriction on the number of reading buffers. This can make sense in a multi-threading context. However programmer should keep in mind efficiency problems if the same records happen to be duplicated several times in memory at the same time.

Associated subroutines are:

- `reallocate_recordbuf`: (re)allocate a record buffer
- `deallocate_recordbuf`: deallocate a record buffer
- `classic_recordbuf_nullify`: force the buffer to forget the object it is used for and the record it duplicates
- `classic_recordbuf_open`: open the buffer for reading a new object

2.8 Miscellaneous

The **CLASSIC** Library provides a convenient type which gathers all the conversion routines from/to the file type to/from the system type, thanks to Fortran 2003 Procedure Pointers. They are evaluated only once, when the file is opened. They greatly help writing generic conversion routines for current and future supported systems (any addition of system should be transparent). The conversion code is also available as **GILDAS** internal values (code 0 means no conversion needed).

```
type :: classic_fileconv_t
  integer(kind=4) :: code   ! RW - Conversion code
  type(subrconv_t) :: read  ! RO - Reading routines (from file to memory)
  type(subrconv_t) :: writ  ! RO - Writing routines (from memory to file)
end type classic_fileconv_t

type :: subrconv_t
  ! Routines for system-data-type <-> file-data-type conversions
  procedure(), nopass, pointer :: i4 => null()  ! RO - for integer*4 values
  procedure(), nopass, pointer :: i8 => null()  ! RO - for integer*8 values
  procedure(), nopass, pointer :: r4 => null()  ! RO - for real*4 values
  procedure(), nopass, pointer :: r8 => null()  ! RO - for real*8 values
  procedure(), nopass, pointer :: cc => null()  ! RO - for character*(*) values
end type subrconv_t
```

Typical calls look like:

```
! Convert 1 INTEGER*4 value from memory to file buffer
call file%conv%writ%i4(i4_memo,i4_file,1)

! Convert 9 contiguous REAL*4 values from file buffer to memory
call file%conv%read%r4(r4_file(:),r4_memo(:),9)
```

For each of the procedure pointers, the 3rd arguments (number of elements to convert) must be an Integer*4 value.

At last but not least, application must initialize the **CLASSIC** Library at startup. It is also advised to *steal* the messages ownership.

- `classic_init`: Initialize the library
- `classic_message_set_id`: Override the owner of the messages coming from the library.

3 Defining and accessing sections

While the content of the sections and the entry index is let (almost) free to the programmer, so that they contain any kind of data on the disk, there are some restrictions which are raised because of the current technical limitations. We detail here the Fortran issues (from the standard or its implementation by the compilers), but they are probably true in a way or another for the other programming languages.

3.1 Technical limitations

In the Figure 1, we see that the **CLASSIC** Library is easily able to read and return to the caller the section as a sequence of untyped bytes (thanks to the various addresses provided by the binary container, it knows in which record and at which position the section is written on the disk), referenced as *section buffer* in the figure. However, once the application has retrieved the bytes in the section, how can it give them their sense?

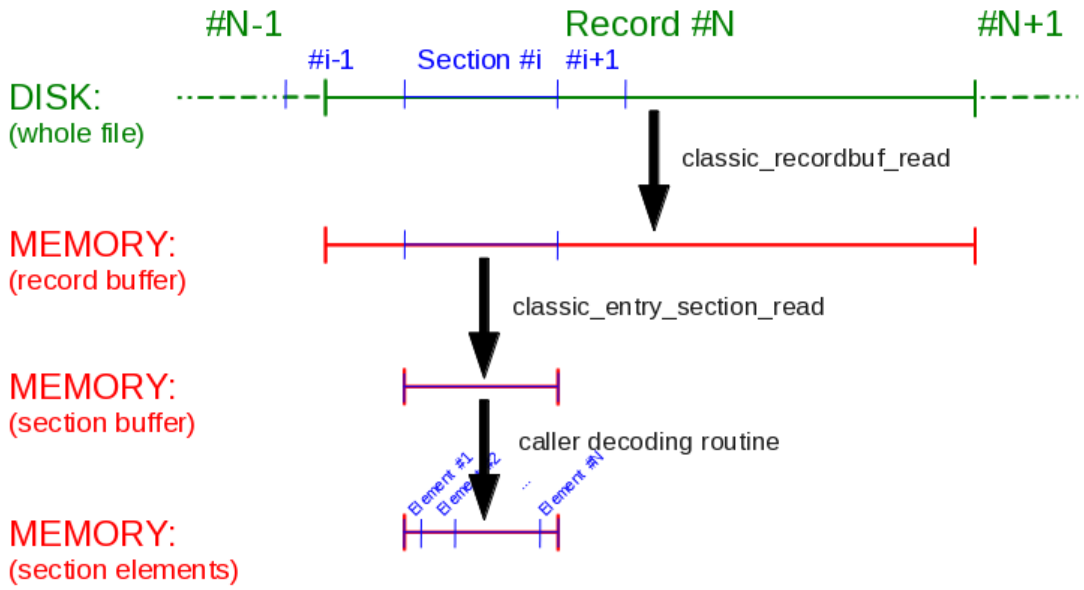

Figure 1: How a section and its elements are read thanks to the **CLASSIC** Library

In an ideal world, one would define an *object* in memory which matches exactly the elements in the section; for example, the *object* could be a contiguous sequence of 1 integer*4, 1 real*4, and 2 integer*8. A bit to bit copy of the bytes into the *object* would be enough to map them into typed variables.

Unfortunately, the Fortran language does not promises such an ideal *object* in all cases. Here is a list of the problems which can be faced:

1. Memory misalignment: neither (the compiler implementations of) the Fortran *sequence derived types*[1] nor the *common blocks* accept misaligned components. For example, gfortran contructs the sequence [I*4, I*8, I*4] in memory as [I*4, 4 unused bytes, I*8, I*4, 4 unused bytes], i.e. it adds an implicit padding so that the elements are aligned on a 64 bits grid. This is not forbidden by the Fortran standard. In this case, we see that a bit to bit copy will not transfer the bytes into the correct components. Ifort, on the other hand, does not add implicit padding, but leaves the components misaligned on the 64 bits grid, which may lead to further problems when reaccessing the data.

2. Character strings: as of the Fortran 2008 standard there is no restriction to put character strings in *sequence derived types* or in *common blocks* (except memory misalignment, see above). However, a character string is a component more complicated than a numeric variable: its length must be described somewhere in memory. This description (kind and position) is a pure compiler choice, which can also lead to unexpected position of the components in memory.

Are there solutions or workarounds to these problems?

1. Memory misalignment can be avoided with 3 strategies:

[1]The Fortran standard provides *derived types* with the `SEQUENCE` statement (so-called *sequence types*), which enforces the ordering of their components (in memory), but does not promise their contiguity. Contiguity is more a concern of the compiler implementation.

- define the whole section on disk so that there will be no misalignment problems in memory. Pro: copy by (full or partial) block will be allowed. Cons: this adds constraints on the types and order of the components on the disk, resulting in a non-intuitive section. This is also error-prone for the programmer, because some compilers do not warn at all when misalignments occur.
- use a (sequenced or not) derived type in memory, and read the components one by one. Pros: definition of the section is completely free. Cons: the ability to read by block is lost, this is less efficient.
- choose an intermediate solution, i.e. define the section so that some parts can be read by block (e.g. gather the 8-bytes words components together, then the 4-bytes words components, then the character strings), and the problematic ones can be read one by one. Pro: copy by partial block will be allowed, minimizes programming errors. Cons: this also adds constraints in the types and may result in non-intuitive sections.

2. Character strings problems can be avoided with the 2 following strategies:

   - define the section in memory so that the bytes (1 byte per character) associated to the string on disk are read into a numeric type (typically scalar or array integer). Pros: can still read in block. Cons: need a second transfer of the bytes from the numeric type to the final character string.
   - assume the memory address of the character string is also the address of its data. That is, that the string length is stored *elsewhere* or *further* and that this does not introduce subsequent alignment problems. As of today, this assumption is correct for ifort and gfortran. Pros: no overlay needed, code clearer and faster. We can reasonably assume this will still work on a mid-term scale. Cons: this is compiler dependent implementation, risk to be a problem at any time.

As of today, the choice made by the applications using the **CLASSIC** Library is:

1. memory misalignments is avoided with the *intermediate* (last) solution exposed above,
2. character strings are read and written as numeric types (first solution), needing one more step for decoding and encoding.

3.2 About byte-swapping

The above description assumes that the bits on disk and in memory are ordered in the same way, i.e. we access a native format. If byte-swapping is needed, bit to bit copies can not be used anymore. In this case the *transfer* of the untyped bytes to the variable depends on the type and kind of the data. However, this still can be done with 2 different strategies:

- transfer the elements one by one,
- transfer the elements by *group*, e.g. make a single subroutine call for 3 contiguous Integer*4, which means the contiguity in memory still provides a benefit.

Depending on the choices made when the data is in native format, the same types and strategies may be used for byte-swapping. One should also take care that byte-swapping *must* be supported, but is encountered very rarely nowadays (only when reading old non-IEEE data).

4 Examples

4.1 Basic file writing

Given the API described in the previous sections, the typical algorithm description to write a **CLASSIC** file can be:

- `call classic_init`
- `call classic_file_init`
- `call classic_filedesc_init`
- `call reallocate_recordbuf` (initialize the working buffers)
- Loop on entries
  - `call classic_entry_init`
  - `call classic_recordbuf_open` (set the working buffer)
  - `call classic_entryindex_write`
  - Loop on sections
    - `call classic_entry_section_add` (add each new section)
  - `call classic_entry_data_add`
  - `call classic_entrydesc_write`
  - `call classic_entry_close`
  - `call classic_filedesc_write` (update the file descriptor)
- `call classic_file_close`
- `call deallocate_recordbuf` (deallocate the working buffers)

A real Fortran program using this algorithm is available in Appendix B.1.

4.2 Basic file reading

The typical algorithm description to read a **CLASSIC** file can be:

- `call classic_init`
- `call classic_file_open`
- `call classic_filedesc_open`
- `call reallocate_recordbuf` (initialize the working buffers)
- Load the File Index for easy browsing. Loop on entries:
  - `call classic_entryindex_read`
- Read the desired entry(ies):
  - `call classic_recordbuf_open` (set the working buffer)
  - `call class_entrydesc_read`
  - `call classic_entrydesc_secfind_all` (find available sections)
  - Loop on available sections

        * `call classic_entry_section_read`
    - `call classic_entry_data_read`

- `call classic_file_close`
- `call deallocate_recordbuf` (deallocate the working buffers)

A real Fortran program using this algorithm is available in Appendix B.2.

A CLASSIC Library detailed API

A.1 File API

```
subroutine classic_file_init(file,version,reclen,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!   Initialize a new Classic file. The following elements must have
! been set before:
!   file%lun
!   file%spec
!   file%nspec
!-----------------------------------------------------------------------
type(classic_file_t), intent(inout) :: file     !
integer(kind=4),      intent(in)    :: version  ! File version
integer(kind=4),      intent(in)    :: reclen   ! Record length (words)
logical,              intent(inout) :: error    ! Logical error flag
end subroutine classic_file_init

subroutine classic_file_open(file,readwrite,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!   Open an old Classic file. The following elements must have been
! set before:
!   file%lun
!   file%spec
!   file%nspec
!-----------------------------------------------------------------------
type(classic_file_t), intent(inout) :: file       !
logical,              intent(in)    :: readwrite  !
logical,              intent(inout) :: error      ! Logical error flag
end subroutine classic_file_open

subroutine classic_file_close(file,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!  Perform the needed operations to close a classic_file_t structure
!-----------------------------------------------------------------------
type(classic_file_t), intent(inout) :: file   !
logical,              intent(inout) :: error  !
end subroutine classic_file_close

subroutine classic_file_loss(file,unused,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
! Parse the input file and search for unused (lost) bytes
!-----------------------------------------------------------------------
type(classic_file_t), intent(inout) :: file    !
integer(kind=4),      intent(in)    :: unused  ! Number of unused bytes per Index
logical,              intent(inout) :: error   !
end subroutine classic_file_loss

subroutine classic_file_copy(in,ou,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!  Copy the input classic_file_t into the output one, taking care
! of some special elements like pointers and allocatables.
!-----------------------------------------------------------------------
type(classic_file_t), intent(in)    :: in     !
type(classic_file_t), intent(inout) :: ou     !
```

```
logical,              intent(inout) :: error  ! Logical error flag
end subroutine classic_file_copy

subroutine classic_file_fopen(file,statu,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Fortran-open the given Classic file. The following elements must
! have been set before:
!   file%lun
!   file%spec
!   file%nspec
!   file%readwrite
!   file%desc%reclen
!---------------------------------------------------------------------
type(classic_file_t), intent(in)    :: file   !
character(len=*),     intent(in)    :: statu  ! 'OLD' or 'NEW'
logical,              intent(inout) :: error  !
end subroutine classic_file_fopen

subroutine classic_file_fclose(file,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Fortran-close the given Classic file. The following elements must
! have been set before:
!   file%lun
!   file%spec
!---------------------------------------------------------------------
type(classic_file_t), intent(in)    :: file   !
logical,              intent(inout) :: error  !
end subroutine classic_file_fclose

subroutine classic_file_fflush(file,error)
use classic_types
!---------------------------------------------------------------------
! @ public
! Fortran-flush the given Classic file. According to Fortran:
!   "Execution of a FLUSH statement causes data written to an
!    external file to be available to other processes, or causes data
!    placed in an external file by means other than Fortran to be
!    available to a READ statement. These actions are processor
!    dependent."
! In other words, it works on both writing and reading files.
!---------------------------------------------------------------------
type(classic_file_t), intent(in)    :: file   !
logical,              intent(inout) :: error  !
end subroutine classic_file_fflush
```

A.2 File Descriptor API

```
subroutine classic_filedesc_init(file,fkind,lsingle,lsize,vind,lind,gex,error)
use classic_params
use classic_types
!---------------------------------------------------------------------
! @ public
!   Initialize a new File Descriptor.
!   The following elements must have been set before:
!       file%lun, file%spec, file%nspec
!   On return, those are set:
!       file%desc, file%conv
!---------------------------------------------------------------------
type(classic_file_t),       intent(inout) :: file     !
integer(kind=4),            intent(in)    :: fkind    ! File kind
logical,                    intent(in)    :: lsingle  ! Single or multiple
```

```
integer(kind=entry_length), intent(in)    :: lsize    ! Total number of entries (linear growth)
integer(kind=4),            intent(in)    :: vind     ! Index version
integer(kind=4),            intent(in)    :: lind     ! Index length (words)
real(kind=4),               intent(in)    :: gex      ! Extension growth
logical,                    intent(inout) :: error    ! Logical error flag
end subroutine classic_filedesc_init

subroutine classic_filedesc_open(file,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
! Open and read the File Descriptor, namely:
! - Get the File Code and set the File-Data-Type-to-System-Data-Type
!   conversion code,
! - Read the File Descriptor from the file, and take care of the
!   possible conversion.
! Fortran-open the file itself if needed.
!-----------------------------------------------------------------------
type(classic_file_t), intent(inout) :: file   !
logical,              intent(out)   :: error  ! Logical error flag
end subroutine classic_filedesc_open

subroutine classic_filedesc_read(file,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!   Read the File Descriptor from the file, and take care of the
! possible conversion. The file itself must have been opened before.
! The conversion code is not reevaluated.
!   It may be useful to re-read a File Descriptor because another
! program might have updated it since last classic_filedesc_open or
! classic_filedesc_read.
!-----------------------------------------------------------------------
type(classic_file_t), intent(inout) :: file   !
logical,              intent(inout) :: error  ! Logical error flag
end subroutine classic_filedesc_read

subroutine classic_filedesc_write(file,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
! Write the File Descriptor to the file.
! The file itself must have been opened before
!-----------------------------------------------------------------------
type(classic_file_t), intent(in)    :: file   !
logical,              intent(inout) :: error  ! Logical error flag
end subroutine classic_filedesc_write

subroutine classic_filedesc_dump(fdesc,name)
use classic_types
!-----------------------------------------------------------------------
! @ public
! Dump the content of the File Descriptor
!-----------------------------------------------------------------------
type(classic_filedesc_t), intent(in) :: fdesc  ! The File Descriptor
character(len=*),         intent(in) :: name   ! The File Descriptor name (some prefix)
end subroutine classic_filedesc_dump
```

A.3 File Index API

```
subroutine classic_entryindex_read(file,entry_num,data,buf,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!  Read an entry in the given file, and return its Entry Index in the
```

```
! 'data(*)' buffer
!-----------------------------------------------------------------------
type(classic_file_t),       intent(in)    :: file       !
integer(kind=entry_length), intent(in)    :: entry_num  ! Entry number
integer(kind=4),            intent(out)   :: data(*)    !
type(classic_recordbuf_t),  intent(inout) :: buf        ! Working buffer
logical,                    intent(inout) :: error      ! Logical errror flag
end subroutine classic_entryindex_read

subroutine classic_entryindex_write(file,entry_num,data,buf,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!   Write the given Entry Index ('data(*)' buffer) for the given entry
! in the given file. There must be room already provisioned for the
! entry in the Extension Index.
!-----------------------------------------------------------------------
type(classic_file_t),       intent(in)    :: file       !
integer(kind=entry_length), intent(in)    :: entry_num  ! Entry number
integer(kind=4),            intent(in)    :: data(*)    !
type(classic_recordbuf_t),  intent(inout) :: buf        ! Working buffer
logical,                    intent(inout) :: error      ! Logical errror flag
end subroutine classic_entryindex_write
```

A.4 Entry API

```
subroutine classic_entry_init(file,xnum,maxsec,version,full,ed,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
! Initialize the Entry for a new observation
!-----------------------------------------------------------------------
type(classic_file_t),       intent(inout) :: file     !
integer(kind=entry_length), intent(in)    :: xnum     ! The corresponding Entry number
integer(kind=4),            intent(in)    :: maxsec   ! Maximum number of sections the ed can describe
integer(kind=4),            intent(in)    :: version  ! Observation version
logical,                    intent(out)   :: full     ! File is full?
type(classic_entrydesc_t),  intent(out)   :: ed       ! The new Entry Descriptor
logical,                    intent(inout) :: error    ! Logical error flag
end subroutine classic_entry_init

subroutine classic_entry_section_read(isec,lsec,sec,ed,buf,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!  Read a section from an entry
!-----------------------------------------------------------------------
integer(kind=4),           intent(in)    :: isec    ! Section id
integer(kind=data_length), intent(inout) :: lsec    ! Length of section (updated on return)
integer(kind=4),           intent(inout) :: sec(*)  ! Section buffer
type(classic_entrydesc_t), intent(in)    :: ed      ! The Entry Descriptor
type(classic_recordbuf_t), intent(inout) :: buf     ! Working buffer
logical,                   intent(inout) :: error   ! Logical error flag
end subroutine classic_entry_section_read

subroutine classic_entry_data_read(array,ndata,ed,buf,error)
use classic_types
!-----------------------------------------------------------------------
! @ public
!  Read the 'data' block from the entry
!-----------------------------------------------------------------------
integer(kind=data_length), intent(inout) :: ndata         ! Length in 32_bit words (updated)
real(kind=4),              intent(out)   :: array(ndata)  ! Data itself
type(classic_entrydesc_t), intent(in)    :: ed            ! The Entry Descriptor
type(classic_recordbuf_t), intent(inout) :: buf           ! Working buffer
```

```
logical,                   intent(inout) :: error         ! Logical error flag
end subroutine classic_entry_data_read

subroutine classic_entry_data_readsub(array,ndata,first,last,ed,buf,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Read a subset of the 'data' block from the entry
!---------------------------------------------------------------------
integer(kind=data_length), intent(inout) :: ndata         ! Length in 32_bit words (updated)
real(kind=4),              intent(out)   :: array(ndata)  ! Data itself
integer(kind=data_length), intent(in)    :: first         ! First data word to read
integer(kind=data_length), intent(in)    :: last          ! Last data word to read
type(classic_entrydesc_t), intent(in)    :: ed            ! The Entry Descriptor
type(classic_recordbuf_t), intent(inout) :: buf           ! Working buffer
logical,                   intent(inout) :: error         ! Logical error flag
end subroutine classic_entry_data_readsub

subroutine classic_entry_section_add(isec,lsec,sec,ed,buf,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Add a new section to an entry. Must not exist before.
!---------------------------------------------------------------------
integer(kind=4),           intent(in)    :: isec    ! Section id
integer(kind=data_length), intent(in)    :: lsec    ! Length of section
integer(kind=4),           intent(in)    :: sec(*)  ! Section buffer
type(classic_entrydesc_t), intent(inout) :: ed      ! The Entry Descriptor
type(classic_recordbuf_t), intent(inout) :: buf     ! Working buffer
logical,                   intent(inout) :: error   ! Logical error flag
end subroutine classic_entry_section_add

subroutine classic_entry_section_update(isec,lsec,sec,ed,buf,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Update an old section in an entry. Must exist before.
!---------------------------------------------------------------------
integer(kind=4),           intent(in)    :: isec    ! Section id
integer(kind=data_length), intent(in)    :: lsec    ! Length of section
integer(kind=4),           intent(in)    :: sec(*)  ! Section buffer
type(classic_entrydesc_t), intent(in)    :: ed      ! The Entry Descriptor
type(classic_recordbuf_t), intent(inout) :: buf     ! Working buffer
logical,                   intent(inout) :: error   ! Logical error flag
end subroutine classic_entry_section_update

subroutine classic_entry_data_add(array,ndata,ed,buf,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Write the 'data' block to the entry. Must not exist before.
!---------------------------------------------------------------------
integer(kind=data_length), intent(in)    :: ndata         ! Length in 32_bit words
real(kind=4),              intent(in)    :: array(ndata)  ! Data itself
type(classic_entrydesc_t), intent(inout) :: ed            ! The Entry Descriptor
type(classic_recordbuf_t), intent(inout) :: buf           ! Working buffer
logical,                   intent(inout) :: error         ! Logical error flag
end subroutine classic_entry_data_add

subroutine classic_entry_data_update(array,ndata,ed,buf,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Update the 'data' block to the entry. Must exist before.
!---------------------------------------------------------------------
```

```
integer(kind=data_length), intent(in)    :: ndata         ! Length in 32_bit words
real(kind=4),              intent(in)    :: array(ndata)  ! Data itself
type(classic_entrydesc_t), intent(inout) :: ed            ! The Entry Descriptor
type(classic_recordbuf_t), intent(inout) :: buf           ! Working buffer
logical,                   intent(inout) :: error         ! Logical error flag
end subroutine classic_entry_data_update

subroutine classic_entry_close(file,buf,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Perform the needed operations when all the entry elements have
! been written.
!---------------------------------------------------------------------
type(classic_file_t),      intent(inout) :: file   !
type(classic_recordbuf_t), intent(in)    :: buf    !
logical,                   intent(inout) :: error  !
end subroutine classic_entry_close
```

A.5 Entry Descriptor API

```
subroutine classic_entrydesc_read(file,buf,ed,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!   Read the Entry Descriptor
!---------------------------------------------------------------------
type(classic_file_t),      intent(in)    :: file   !
type(classic_recordbuf_t), intent(inout) :: buf    ! Working buffer
type(classic_entrydesc_t), intent(inout) :: ed     ! The Entry Descriptor read
logical,                   intent(inout) :: error  ! Logical error flag
end subroutine classic_entrydesc_read

subroutine classic_entrydesc_write(file,buf,ed,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!   Write the input Entry Descriptor
!---------------------------------------------------------------------
type(classic_file_t),      intent(inout) :: file   !
type(classic_recordbuf_t), intent(inout) :: buf    ! Working buffer
type(classic_entrydesc_t), intent(in)    :: ed     ! The Entry Descriptor to be written
logical,                   intent(inout) :: error  ! Logical error flag
end subroutine classic_entrydesc_write

subroutine classic_entrydesc_secfind_one(ed,iden,found,num)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Find if 1 section is present in the input Entry Descriptor.
! Inefficient if you test each section one by one.
!---------------------------------------------------------------------
type(classic_entrydesc_t), intent(in)  :: ed     ! The Entry Descriptor
integer(kind=4),           intent(in)  :: iden   ! The section identifier
logical,                   intent(out) :: found  ! Section was found or not?
integer(kind=4),           intent(out) :: num    ! The section number (if found)
end subroutine classic_entrydesc_secfind_one

subroutine classic_entrydesc_secfind_all(ed,found,first,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!   Fill a 'found' array for all sections present or not. This assumes
! that the section identifiers (seciden(:) values) can be used as
! position index (maybe with an offset) in the 'found' array.
```

```
!---------------------------------------------------------------------
type(classic_entrydesc_t), intent(in)    :: ed        ! The Entry Descriptor
logical,                   intent(out)   :: found(:)  ! Sections were found or not?
integer(kind=4),           intent(in)    :: first     ! First index in the array (offset)
logical,                   intent(inout) :: error     ! Logical error flag
end subroutine classic_entrydesc_secfind_all

subroutine classic_entrydesc_dump(ed)
use gbl_message
use classic_types
!---------------------------------------------------------------------
! @ public
!  Dump the input Entry Descriptor
!---------------------------------------------------------------------
type(classic_entrydesc_t), intent(in) :: ed  ! The Entry Descriptor
end subroutine classic_entrydesc_dump
```

A.6 Buffer API

```
subroutine reallocate_recordbuf(buf,len,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Allocate or reallocate the input working buffer
!---------------------------------------------------------------------
type(classic_recordbuf_t), intent(inout) :: buf    !
integer(kind=4),           intent(in)    :: len    ! New length required
logical,                   intent(inout) :: error  ! Logical error flag
end subroutine reallocate_recordbuf

subroutine deallocate_recordbuf(buf,error)
use classic_types
!---------------------------------------------------------------------
! @ public
!  Deallocate the input working buffer
!---------------------------------------------------------------------
type(classic_recordbuf_t), intent(inout) :: buf    !
logical,                   intent(inout) :: error  ! Logical error flag
end subroutine deallocate_recordbuf

subroutine classic_recordbuf_nullify(buf)
use classic_types
!---------------------------------------------------------------------
! @ public
!  "Nullify" the input buffer i.e. reset its record positions so that
! recordbuf_read/write won't assume something useful is stored in the
! data buffer.
!---------------------------------------------------------------------
type(classic_recordbuf_t), intent(inout) :: buf  !
end subroutine classic_recordbuf_nullify

subroutine classic_recordbuf_open(file,rstart,wstart,buf,error)
use classic_types
!---------------------------------------------------------------------
! @ public
! Open the buffer for reading a new object
!---------------------------------------------------------------------
type(classic_file_t),      intent(in)    :: file    !
integer(kind=8),           intent(in)    :: rstart  ! First record of the object
integer(kind=4),           intent(in)    :: wstart  ! First word in this record
type(classic_recordbuf_t), intent(inout) :: buf     !
logical,                   intent(inout) :: error   !
end subroutine classic_recordbuf_open
```

A.7 Miscellaneous

```
subroutine classic_init(error)
use classic_vars
!-----------------------------------------------------------------------
! @ public
!  Initialize the Classic library. This should be done only once at
! startup. The library will refuse opening a Classic file as long
! as this has not been done.
!-----------------------------------------------------------------------
logical, intent(inout) :: error  ! Logical error flag
end subroutine classic_init

subroutine classic_message_set_id(id)
!-----------------------------------------------------------------------
! @ public
! Alter library id into input id. Should be called by the library
! which wants to share its id with the current one.
!-----------------------------------------------------------------------
integer(kind=4), intent(in) :: id         !
end subroutine classic_message_set_id
```

B Examples

B.1 File writing

```
program classic_demo_write
  use classic_api
  implicit none
  !---------------------------------------------------------------------
  !  Demonstration program which writes a Classic file version 2, with
  ! a few entries, and 1 Entry Index, 1 section, and 1 data array per
  ! entry. The output file 'foo.bin' can be read again by the reading
  ! demonstration program.
  !
  !  * Entry Index:
  !     - Entry starting record  (I*8)
  !     - Entry starting word    (I*4)
  !     - Dummy 'foo' value      (I*4)
  !  * Section:
  !     - One R*4 value
  !     - One R*8 value
  !     - One I*4 value
  !     - One I*8 value
  !     - One C*8 value
  !  * Data:
  !     - Array of 10 R*4 values
  !---------------------------------------------------------------------
  type(classic_recordbuf_t) :: rbufind,rbufobs
  integer(kind=4) :: i
  integer(kind=entry_length) :: ientry
  logical :: error,full
  ! File
  type(classic_file_t) :: file
  integer(kind=4), parameter :: file_lun=50  ! Arbitrary value
  character(len=*), parameter :: file_name='foo.bin'
  integer(kind=4), parameter :: file_version=2
  integer(kind=4), parameter :: file_reclen=64  ! Arbitrary value
  ! File Descriptor
  integer(kind=4), parameter :: fdesc_kind=classic_kind_demo
  logical, parameter :: fdesc_single=.true.
  real(kind=4), parameter :: fdesc_expo=2.0
  integer(kind=entry_length), parameter :: fdesc_size=1  ! Not relevant for exponential growth
  ! Entry Index
  integer(kind=4), parameter :: ind_version=123  ! Index version (arbitrary value)
  integer(kind=4), parameter :: ind_length=4  ! 1 I*8, 2 I*4
  integer(kind=8) :: ind_rstart
  integer(kind=4) :: ind_wstart
  integer(kind=4) :: ind_foo
  integer(kind=4) :: bufind(ind_length)  ! Buffer for byte-swapping one index
  ! Entry Descriptor
  type(classic_entrydesc_t) :: edesc
  integer(kind=4), parameter :: edesc_maxsec=1     ! Maximum number of section per entry
  integer(kind=4), parameter :: edesc_version=456  ! Observation version (arbitrary value)
  ! Section
  integer(kind=4), parameter :: sec_id=789  ! Identifier (arbitrary value)
  integer(kind=data_length), parameter :: sec_length=8  ! 1 R*4, 1 R*8, 1 I*4, 1 I*8, 1 C*8
  real(kind=4) :: sec_r4
  real(kind=8) :: sec_r8
  integer(kind=4) :: sec_i4
  integer(kind=8) :: sec_i8
  character(len=8) :: sec_c8
  integer(kind=4) :: bufsec(sec_length)  ! Buffer for byte-swapping one section
  ! Data
  integer(kind=data_length), parameter :: dat_length=10
  real(kind=4) :: data(dat_length)
  real(kind=4) :: bufdat(dat_length)  ! Buffer for byte-swapping one data block
```

```
!
write(*,'(A)')  '--- Welcome to the Classic demonstration writing program ---'
!
error = .false.
!
! Initialize the Classic library
call classic_init(error)
if (error)  stop
!
file%lun = file_lun
file%spec = file_name
file%nspec = len_trim(file%spec)
file%update = .false.
call classic_file_init(file,file_version,file_reclen,error)
if (error)  stop
!
call classic_filedesc_init(file,fdesc_kind,fdesc_single,fdesc_size,  &
  ind_version,ind_length,fdesc_expo,error)
if (error)  stop
!
call reallocate_recordbuf(rbufind,file%desc%reclen,error)
if (error)  stop
call reallocate_recordbuf(rbufobs,file%desc%reclen,error)
if (error)  stop
!
do ientry=1,5  ! Write 5 entries
  write(*,*) ''
  write(*,'(A,I0)') 'Writing entry #',ientry
  call classic_entry_init(file,ientry,edesc_maxsec,edesc_version,full,  &
    edesc,error)
  if (full)  write(*,*) 'Error: file is full'
  if (error)  stop
  !
  ! Set the working buffer
  call classic_recordbuf_open(file,file%desc%nextrec,file%desc%nextword,  &
    rbufobs,error)
  if (error)  stop
  !
  ! Add the Entry Index
  ind_rstart = rbufobs%rstart
  ind_wstart = rbufobs%wstart
  ind_foo = 900+ientry  ! Some dummy value to be stored in the Index
  write(*,'(A,I0,1X,I0,1X,I0)')  &
    '  INDEX: ',ind_rstart,ind_wstart,ind_foo
  ! Entry Index can be anything => byte swapping is application specific.
  call classic_demo_ind2buf(ind_rstart,ind_wstart,ind_foo,bufind,  &
    file%conv)
  if (error)  stop
  call classic_entryindex_write(file,ientry,bufind,rbufind,error)
  if (error)  stop
  !
  ! Add 1 section: 1 R*4, 1 R*8, 1 I*4, 1 I*8, 1 C*8
  sec_r4 = real(ientry,kind=4)
  sec_r8 = real(ientry,kind=8)
  sec_i4 = int(ientry,kind=4)
  sec_i8 = int(ientry,kind=8)
  write(sec_c8,'(I8)')  ientry
  write(*,'(A,F0.2,1X,F0.4,1X,I0,1X,I0,1X,A)')   &
    '  SECTION: ',sec_r4,sec_r8,sec_i4,sec_i8,sec_c8
  ! Section can be anything => byte swapping is application specific.
  call classic_demo_sec2buf(sec_r4,sec_r8,sec_i4,sec_i8,sec_c8,bufsec,  &
    file%conv)
  call classic_entry_section_add(sec_id,sec_length,bufsec,edesc,rbufobs,  &
    error)
  if (error)  stop
```

```
    !
    ! Add data
    ! Data can be anything => byte swapping is application specific.
    data(:) = (/ (ientry+i, i=1,dat_length) /)
    write(*,'(A,I0,A,10(1X,F0.2))')  &
      '  DATA (',dat_length,' values): ',data(1:dat_length)
    call classic_demo_dat2buf(int(dat_length,kind=4),data,bufdat,file%conv)
    call classic_entry_data_add(bufdat,dat_length,edesc,rbufobs,error)
    if (error)  stop
    !
    ! Write the Entry Descriptor
    write(*,'(A,I0,1X,I0,1X,I0,1X,I0)')  &
      '  DESCRIPTOR: ',edesc%nsec,edesc%nword,edesc%adata,edesc%ldata
    call classic_entrydesc_write(file,rbufobs,edesc,error)
    if (error)  stop
    !
    call classic_entry_close(file,rbufobs,error)
    if (error)  stop
    !
    ! Once the Entry and Entry Descriptor are written, update the File
    ! Descriptor on disk (in particular xnext)
    file%desc%xnext = ientry+1
    call classic_filedesc_write(file,error)
    if (error)  stop
  enddo
  !
  ! Debugging printout
  write(*,*) ''
  write(*,'(A)')  'File Descriptor:'
  call classic_filedesc_dump(file%desc,'w')
  !
  call classic_file_close(file,error)
  if (error)  stop
  !
  call deallocate_recordbuf(rbufobs,error)
  if (error)  stop
  call deallocate_recordbuf(rbufind,error)
  if (error)  stop
  !
  write(*,*) ''
  write(*,'(A,A,A)') 'File ',trim(file%spec),' successfully written'
  !
contains
  !
  subroutine classic_demo_ind2buf(rec,word,foo,buf,conv)
    use gildas_def
    use gkernel_interfaces
    use classic_api
    !---------------------------------------------------------------------
    ! Copy the Entry Index into the given buffer, swap bytes if needed
    !---------------------------------------------------------------------
    integer(kind=8),          intent(in)  :: rec
    integer(kind=4),          intent(in)  :: word
    integer(kind=4),          intent(in)  :: foo
    integer(kind=4),          intent(out) :: buf(*)
    type(classic_fileconv_t), intent(in)  :: conv
    !
    call conv%writ%i8(rec, buf(1),1)  ! 1-2: record
    call conv%writ%i4(word,buf(3),1)  !   3: word
    call conv%writ%i4(foo, buf(4),1)  !   4: foo
    !
  end subroutine classic_demo_ind2buf
  !
  subroutine classic_demo_sec2buf(r4,r8,i4,i8,c8,buf,conv)
    use gildas_def
```

```
    use gkernel_interfaces
    use classic_api
    !---------------------------------------------------------------------
    ! Copy the section values into the given buffer, swap bytes if needed
    !---------------------------------------------------------------------
    real(kind=4),             intent(in)  :: r4
    real(kind=8),             intent(in)  :: r8
    integer(kind=4),          intent(in)  :: i4
    integer(kind=8),          intent(in)  :: i8
    character(len=*),         intent(in)  :: c8
    integer(kind=4),          intent(out) :: buf(*)
    type(classic_fileconv_t), intent(in)  :: conv
    !
    call conv%writ%r4(r4,buf(1),1)  !   1: R*4
    call conv%writ%r8(r8,buf(2),1)  ! 2-3: R*8
    call conv%writ%i4(i4,buf(4),1)  !   4: I*4
    call conv%writ%i8(i8,buf(5),1)  ! 5-6: I*8
    call conv%writ%cc(c8,buf(7),2)  ! 7-8: C*8
    !
  end subroutine classic_demo_sec2buf
  !
  subroutine classic_demo_dat2buf(ldata,data,buf,conv)
    use gildas_def
    use gkernel_interfaces
    use classic_api
    !---------------------------------------------------------------------
    ! Copy the Data array into the given buffer, swap bytes if needed
    !---------------------------------------------------------------------
    integer(kind=4),          intent(in)  :: ldata
    real(kind=4),             intent(in)  :: data(ldata)
    real(kind=4),             intent(out) :: buf(ldata)
    type(classic_fileconv_t), intent(in)  :: conv
    !
    call conv%writ%r4(data,buf,ldata)
    !
  end subroutine classic_demo_dat2buf
  !
end program classic_demo_write
```

B.2 File reading

```
program classic_demo_read
  use classic_api
  implicit none
  !---------------------------------------------------------------------
  !  Demonstration program which reads a Classic file version 2, with a
  ! few entries, and 1 Entry Index, 1 section, and 1 data array per
  ! entry. The input file 'foo.bin' should have created first with the
  ! writing demonstration program.
  !
  !  * Entry Index:
  !     - Entry starting record  (I*8)
  !     - Entry starting word    (I*4)
  !     - Dummy 'foo' value      (I*4)
  !  * Section:
  !     - One R*4 value
  !     - One R*8 value
  !     - One I*4 value
  !     - One I*8 value
  !     - One C*8 value
  !  * Data:
  !     - Array of 10 R*4 values
  !---------------------------------------------------------------------
  type(classic_recordbuf_t) :: rbufind,rbufobs
  integer(kind=entry_length) :: ientry
```

```
logical :: error
! File
type(classic_file_t) :: file
integer(kind=4), parameter :: file_lun=50  ! Arbitrary value
character(len=*), parameter :: file_name='foo.bin'
logical, parameter :: file_readwrite=.false.  ! Open the file read-only
! File Descriptor
integer(kind=4), parameter :: fdesc_kind=classic_kind_demo
! Entry Index
integer(kind=4), parameter :: ind_version=123  ! Index version (arbitrary value)
integer(kind=4), parameter :: ind_length=4  ! 1 I*8, 2 I*4
integer(kind=8) :: ind_rstart
integer(kind=4) :: ind_wstart
integer(kind=4) :: ind_foo
integer(kind=4) :: bufind(ind_length)  ! Buffer for byte-swapping one index
! Entry Descriptor
type(classic_entrydesc_t) :: edesc
integer(kind=4), parameter :: edesc_version=456  ! Observation version (arbitrary value)
! Section
integer(kind=4), parameter :: sec_id=789  ! Identifier (arbitrary value)
integer(kind=data_length), parameter :: sec_length_max=8  ! Max section length
integer(kind=data_length) :: sec_length                   ! Actual section length
real(kind=4) :: sec_r4
real(kind=8) :: sec_r8
integer(kind=4) :: sec_i4
integer(kind=8) :: sec_i8
character(len=8) :: sec_c8
integer(kind=4) :: bufsec(sec_length_max)  ! Buffer for byte-swapping one section
! Data
integer(kind=data_length), parameter :: dat_length_max=10  ! Max data length
integer(kind=data_length) :: dat_length                    ! Actual data length
real(kind=4) :: data(dat_length_max)
real(kind=4) :: bufdat(dat_length_max)  ! Buffer for byte-swapping one data block
!
write(*,'(A)')  '--- Welcome to the Classic demonstration reading program ---'
!
error = .false.
!
! Initialize the Classic library
call classic_init(error)
if (error)  stop
!
file%lun = file_lun
file%spec = file_name
file%nspec = len_trim(file%spec)
file%update = .false.
call classic_file_open(file,file_readwrite,error)
if (error)  stop
!
call classic_filedesc_open(file,error)
if (error)  stop
write(*,*) ''
write(*,'(A)')  'File Descriptor:'
call classic_filedesc_dump(file%desc,'r')
!
if (file%desc%kind.ne.fdesc_kind) then
  write(*,*) 'Error: unexpected file kind'
  stop
elseif (file%desc%vind.ne.ind_version) then
  write(*,*) 'Error: unsupported index version'
  stop
endif
!
call reallocate_recordbuf(rbufind,file%desc%reclen,error)
if (error)  stop
```

```
  call reallocate_recordbuf(rbufobs,file%desc%reclen,error)
  if (error)  stop
  !
  do ientry=1,file%desc%xnext-1  ! Read all entries
    write(*,*) ''
    write(*,'(A,I0)') 'Reading entry #',ientry
    !
    ! Read the Entry Index
    call classic_entryindex_read(file,ientry,bufind,rbufind,error)
    if (error)  stop
    ! Entry Index can be anything => byte swapping is application specific
    call classic_demo_buf2ind(bufind,ind_rstart,ind_wstart,ind_foo,file%conv)
    if (error)  stop
    write(*,'(A,I0,1X,I0,1X,I0)')  &
      '  INDEX: ',ind_rstart,ind_wstart,ind_foo
    !
    ! Set the working buffer
    call classic_recordbuf_open(file,ind_rstart,ind_wstart,rbufobs,error)
    if (error)  stop
    !
    ! Read the Entry Descriptor
    call classic_entrydesc_read(file,rbufobs,edesc,error)
    if (error)  stop
    write(*,'(A,I0,1X,I0,1X,I0,1X,I0)')  &
      '  DESCRIPTOR: ',edesc%nsec,edesc%nword,edesc%adata,edesc%ldata
    if (edesc%version.ne.edesc_version) then
      write(*,*) 'Error: unsupported observation version'
      stop
    endif
    !
    ! Read section by identifier
    sec_length = sec_length_max  ! Updated in return
    call classic_entry_section_read(sec_id,sec_length,bufsec,edesc,rbufobs,  &
      error)
    if (error)  stop
    ! Section can be anything => byte swapping is application specific
    call classic_demo_buf2sec(bufsec,sec_r4,sec_r8,sec_i4,sec_i8,sec_c8,  &
      file%conv)
    write(*,'(A,F0.2,1X,F0.4,1X,I0,1X,I0,1X,A)')   &
      '  SECTION: ',sec_r4,sec_r8,sec_i4,sec_i8,sec_c8
    !
    ! Data
    dat_length = dat_length_max  ! Updated in return
    call classic_entry_data_read(bufdat,dat_length,edesc,rbufobs,error)
    if (error)  stop
    call classic_demo_buf2dat(int(dat_length,kind=4),bufdat,data,file%conv)
    write(*,'(A,I0,A,10(1X,F0.2))')  &
      '  DATA (',dat_length,' values): ',data(1:dat_length)
    !
  enddo
  !
  call classic_file_close(file,error)
  if (error)  stop
  !
  call deallocate_recordbuf(rbufobs,error)
  if (error)  stop
  call deallocate_recordbuf(rbufind,error)
  if (error)  stop
  !
  write(*,*) ''
  write(*,'(A,A,A)') 'File ',trim(file%spec),' successfully read'
  !
contains
  !
  subroutine classic_demo_buf2ind(buf,rec,word,foo,conv)
```

```
    use gildas_def
    use gkernel_interfaces
    use classic_api
    !------------------------------------------------------------------
    ! Copy the Entry Index into the given buffer, swap bytes if needed
    !------------------------------------------------------------------
    integer(kind=4),          intent(in)  :: buf(*)
    integer(kind=8),          intent(out) :: rec
    integer(kind=4),          intent(out) :: word
    integer(kind=4),          intent(out) :: foo
    type(classic_fileconv_t), intent(in)  :: conv
    !
    call conv%read%i8(buf(1),rec, 1)  ! 1-2: record
    call conv%read%i4(buf(3),word,1)  !   3: word
    call conv%read%i4(buf(4),foo, 1)  !   4: foo
    !
  end subroutine classic_demo_buf2ind
  !
  subroutine classic_demo_buf2sec(buf,r4,r8,i4,i8,c8,conv)
    use gildas_def
    use gkernel_interfaces
    use classic_api
    !------------------------------------------------------------------
    ! Copy the section values into the given buffer, swap bytes if needed
    !------------------------------------------------------------------
    integer(kind=4),          intent(in)  :: buf(*)
    real(kind=4),             intent(out) :: r4
    real(kind=8),             intent(out) :: r8
    integer(kind=4),          intent(out) :: i4
    integer(kind=8),          intent(out) :: i8
    character(len=*),         intent(out) :: c8
    type(classic_fileconv_t), intent(in)  :: conv
    !
    call conv%read%r4(buf(1),r4,1)  !   1: R*4
    call conv%read%r8(buf(2),r8,1)  ! 2-3: R*8
    call conv%read%i4(buf(4),i4,1)  !   4: I*4
    call conv%read%i8(buf(5),i8,1)  ! 5-6: I*8
    call conv%read%cc(buf(7),c8,2)  ! 7-8: C*8
    !
  end subroutine classic_demo_buf2sec
  !
  subroutine classic_demo_buf2dat(ldata,buf,data,conv)
    use gildas_def
    use gkernel_interfaces
    use classic_api
    !------------------------------------------------------------------
    ! Copy the Data array into the given buffer, swap bytes if needed
    !------------------------------------------------------------------
    integer(kind=4),          intent(in)  :: ldata
    real(kind=4),             intent(in)  :: buf(ldata)
    real(kind=4),             intent(out) :: data(ldata)
    type(classic_fileconv_t), intent(in)  :: conv
    !
    call conv%read%r4(buf,data,ldata)
    !
  end subroutine classic_demo_buf2dat
  !
end program classic_demo_read
```